OpenMP

→ directive based API

→ canbe used with C, C++, FORTRAN for programming Shared address spacemachines

→ It's a library

→ These directives provides support for concurrency, synchronization and data handling without the need to explicitly setup the mutexes, conditional variable, data scope and initilization.

→ when we use openMP in C, C++, then its based on # pragma compiler directives

→ A directive consists of a directive name followed by clauses.

pragma omp directive [clause list]

→ These execute serially until and unless they encounter parallel directive, which create a no. of threads.

pragma omp parallel [clause list]

→ the clause list is used to specify conditional parallelization, no. of threads, data handling.

Conditional parallelization:- the if clause determing whether the parallel construct has to spin up threads or not.

Degree of Concurrency :- The num_threads (int-exp) clause specifies the no. of threads to be created / are created.

Data handling :- clauses

(i) private (variable list) → indicates variables local to each thread.

(ii) first private (variable list) → similar to private, but the variables are ~~assigned~~ assigned values before the parallel directive is encountered

(iii) shared (variable list) → indicates that variables are shared ~~accr~~ across all the threads.

→ to run an ~~omp~~ openMP program, we have to have

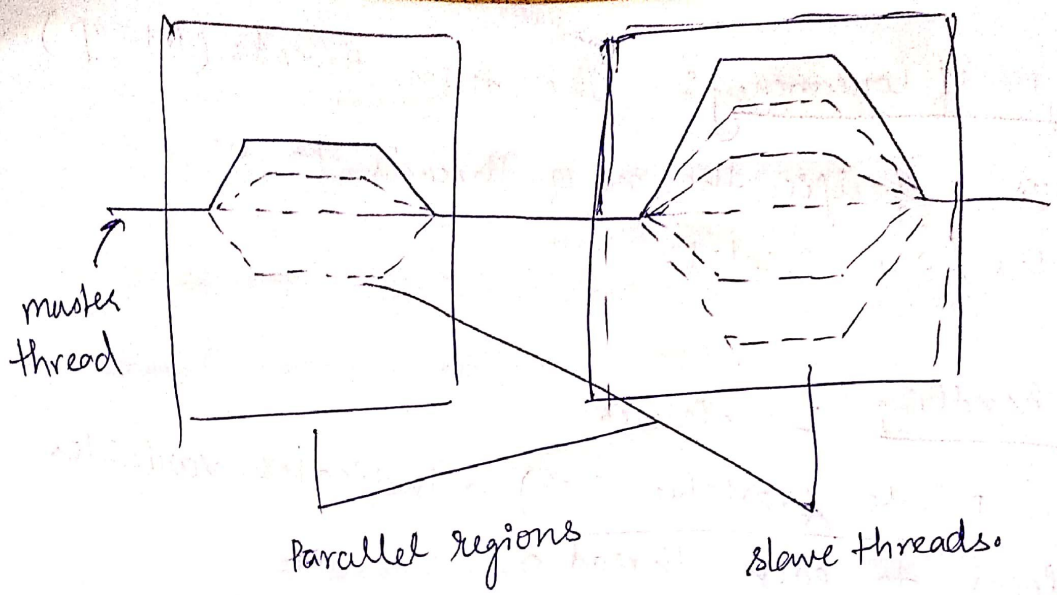
(i) # include <omp.h>

(ii) to compile " gcc -fopenmp filename.c "

OPENMP : Programming Model

→ Fork-Join Parallelism :- Master threads generates a team of threads as needed.

* There will be a master thread executing the serial code, but when required at some point, it spins up new threads (slaves) to work in parallel and again merge back and continue to do serial execution.



Reduction clause in OpenMP

→ It specifies how multiple ~~threads sharing~~ local copies of a variable at different threads are combined into a single copy at the master when thread exits.

→ Prototype ⇒ reduction (operator : variable list)

→ the operator can be +, -, x, ÷, |, ^, &&, ||

Specifying Concurrent tasks

→ the "parallel" directive can be used in conjunction with other directives to acheive concurrency accross iterations and tasks.

→ 'for' and 'sections' directives are provided by openMP to specify concurrent tasks.

→ The 'for' directive is used to split parallel iteration spaces accross threads

→ prototype → # pragma omp for [clause list]

→ By dfault there is a barriers at the end of the "omp for"

↓

It's the point at which all the threads should stop and can't proceed further until all other threads reach this barrier or point

→ to turnoff (disable) the barrier, 'nowait' clause can be used.

1 - 250 T_1
250 - 500 T_2
500 - 750 T_3
750 - 1000 T_4

→ Sections directive

- Prototype

# pragma omp sections [clause list]
{
[# pragma omp section
]
[# pragma omp section
]
:
}

Iteration Assignments to threads

→ 'Schedule' clause of the 'for' directive deals with the assignment of iterations to threads.

→ Prototype :

Schedule (Scheduling-class [, parameter])

→ open mp supports four scheduling classes :-

(i) static :- → The iteration space is broken into chunks / blocks of approximately same size.

→ Then N/num threads are given the broken chuncks in round-robin fashion

characteristics :- (i) low overhead
(ii) Good locality
(iii) load imbalance problems occur

(ii) dynamic :- → Threads dynamically grab chunks of N iterations until all iterations are over. if no chunk is specified, N=1

prototype :- Schedule (dynamic, 3)

(iii) guided :- It's similar to dynamic, but the size of chunks decreases as threads grab iterations, but it's at least of size N, if no chunk is specified N=1

characteristics = (i) high overhead
(ii) not very good locality
(iii) can solve imbalance problem

(iv) Runtime :- The scheduling decision comes during the runtime of the program

Synchronization

High level synchronization

→ critical
→ Atomic
→ Barrier
→ ordered

Low level synchronization

→ flush
→ locks (both simple and nested)

Barrier → all threads should arrive at the same point till they proceed for further executions.

eg :-

```
# pragma omp parallel
{
    int id = omp_get_thread_num();
    A[id] = big computation (id);
# pragma omp barrier
    B[id] = big_cal2 (id, A);
}
```

from the example we know that for the B array to be filled up we need all the content present in the Array A, inorder to do that we place a #pragma omp barrier directive as threads waits for all the other threads to complete their part and then proceed for further processing.

- Critical → It allows only one thread at a time to access the critical section (block of code)
  - → It does so by mutual exclusion.
  - → If one thread is inside the block, others have to wait for their turn.

→ prototype #pragma omp critical [(name)]

This can be used to identify a critical region.

→ the use of name allows different threads to executed different code while being protected from each other.

→ name is optional, if not specified, the section is mapped to a default name.

```
#pragma omp parallel sections
{
    #pragma omp section
    {   // consum producer task (thread)
        task = produce_task();
    }
    #pragma omp critical (task_queue)
    {
        insert_into_queue (task);
    }
```

~~# pragma omp parallel sections~~

{

pragma omp section

{

// consumer thread.

pragma omp critical (task_queue)

{

task = extract_from_queue (task);

}

consume_task(task);

}

}

Atomic :- → It's used to do memory updates in an atomic manner. (simple updates like increment, decrement, etc)

→ the update can be of the form, x++, x--, x+= temp, etc

example :-

```
# pragma omp parallel
{
    double tmp, B;
    B = doit();
    tmp = big_ugly(B);
    # pragma omp atomic
        x += tmp;
}
```

Ordered → enclosed code is executed in same order as could occur in sequential execution of the loop.

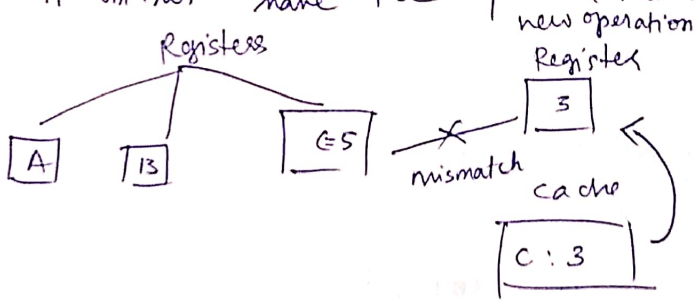
→ Some time we don't want things to be executed in parallel and we want it to executed sequentially, then we can use this directive.

→ # pragma omp ordered

Memory consistency : flush

→ System must provide a consistent view of memory

(i.e) consider a situation where we have to values a and b in registers, adding them both will give us c and we store it in some other register. now if we don't update the c's value in the lower level storage areas, the next time we ask for c it will not have the updated value.



→ Therefore all thread variables must be written back to memory.

→ Implicit flushes happen automatically at

(i) all explicit and implicit barriers

(ii) entry and exit of critical sections

(iii) entry and exist from lock routines.

Data handling in openMP

→ #pragma omp parallel [data scope clauses]

- → Shared
- → private
- → firstprivate
- → lastprivate
- → thread private
- → default
- → reduction.

→ shared

→ the variables are shared among the threads.

→ data corruption is possible when multiple threads try to updat the same memory location.

→ therefore data correctness is user's responsibility

```
Ex:- int x = 1;
     #pragma omp parallel shared (x) num_threads(2)
     {
         x++;
1 —— printf ( " %d " , x);
     }
2 —— printf ( " %d ", x );
```

Output:-
2 (one thread updates it to 2) — 1st printf
3 (other thread updates it to 3) — 1st printf
3 (x value out of the parallel section) — 2nd printf

private :- The variable is private to each and every thread. (i.e) each and every thread will have a ~~local copy of the~~ the variable in their local memory with value assigned to be 0

for example :-

```
int x = 1;
# pragma omp parallel private(x) num_threads(2)
{
    x++;
1 —— printf("%d", x);
}
2 —— Printf("%d", x);
```

output : 1 → (1st thread updates ~~it~~ it to 1 as it has a local copy of x=0 → x=1)
1 → (2nd thread updates it to 1 as it has a local copy of x=0 → x=1)
1 → print outside of the parallel section.

firstprivate :- The variable is private to each and every thread, but here the catch is that the local copy of the variable is initialized with value declared outside of the structured block.

for example :-

```
int x=1;
#pragma omp parallel firstprivate (x) num_threads(2)
{
    x++;
    printf("%d", x);
}
Printf("%d", x);
```

output :-

2
2
1

lastprivate → It transfers the values from parallel region to the outside context.

Reduction → It specifies how multiple local copies of a variable at different threads are combined into a single copy at the master thread when thread exits.

Prototype :- reduction (operator : variable list)

the operator can be +, -, *, %, etc.

threadprivate :- It specifies that variables are replicated, with each thread having its own copy.

→ #pragma omp threadprivate (list)

→ copyin clause :- It provides a mechanism to copy the value of a threadprivate variable of the master thread to the threadprivate variable of each other member of the team.

→ hence the original value of A is retained.

default :- It's used to set the nature of data-sharing of each and every variable in the list

→ #pragma omp default (private(x,y))

OpenMP library functions

Thread Control

void omp_set_num_threads(int num) → set the no of threads to be used in parallel region

int omp_get_thread_num() → get the thread Id from the team

~~int omp_get_thread_n~~

int omp_get_num_threads() → get the no of threads in the team

int omp_get_max_threads() → return the max no of threads that could be used to form a new team

Processor level

int omp_get_num_procs( ) ← gets the no. of processors available on device

Check if inside a parallel region

int omp_in_parallel( ) → if the execution is inside parallel region return true else false

Schedule control

void omp_set_schedule(omp_sched_t kind, int chunk_size)

Void omp_get_schedule(omp_sched_t * kind, int* chunk_size)

Lock routines

Void omp_init_lock(omp_lock_t * mutex) → intilization of lock to unlocked state must be called before the lock can be used.

Void omp_destroy_lock(omp_lock_t* mutex) → to destroy the lock.

Void omp_set_lock(omp_lock_t* mutex) → to set lock

Void omp_unset_lock(omp_lock_t* mutex) → to unset lock / to unlock.

Environment Variables in openMP

→ OMP NUM THREADS → default no. of threads created upon entering a parallel region

→ OMP SET DYNAMIC : determines the no. of threads to changed dynamically

→ OMP SCHEDULE :- scheduling of for loops

MPI (Message Passing Interface)

→ Standardize message passing library specification

→ It's used for parallel computers, clusters and heterogenous networks.

→ It's not a compiler specification

→ many implementations are available like OPEN MPI, MPICH

→ portable with C/C++ / FORTRAN

→ to communicate together in mpi - processes or processors are given ranks (i.e) 0 — k-1 where k no. of processors are there in the communication network.

→ a process is a program performing task on a processor.

SPMD

Single program, multiple dataset

→ same program is run on different processors with different datasets

MPMD

multiple program, multiple dataset

→ different programs are run on different processor with different datasets.

MPI allows SPMD as well as MPMD

Message Passing

→ messages are packets of data moving b/w sub programs.